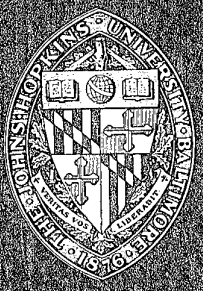
PROCEEDINGS
of
The Seventeenth Annual Conference on Information Sciences and Systems



Department of Electrical Engineering and Computer Science
The Johns Hopkins University
Baltimore, Maryland 21218

P R O C E E D I N G S

O F

T H E S E V E N T E E N T H A N N U A L

C O N F E R E N C E

O N

I N F O R M A T I O N S C I E N C E S A N D S Y S T E M S

PAPERS PRESENTED

March 23, 24 and 25, 1983

PROGRAM DIRECTORS

Howard L. Weinert

Robert C. Melville

Department of Electrical Engineering and Computer Science

The Johns Hopkins University

Baltimore, Maryland 21218

TABLE-DRIVEN TOP-DOWN PARSERS WITH AUTOMATIC ERROR RECOVERY

Henrique M.G. de Aguiar and Michael A. Stanton
Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
22453 - Rio de Janeiro - RJ, Brazil

ABSTRACT

A.C.Hartmann's automatic error recovery method for recursive descent parsers is adapted for use in a table-driven top-down parser based on syntax graphs. The automatic generation of table-driven parsers with error recovery is shown to be possible given only the grammar as input.

1. INTRODUCTION

In his book [HAR77] Hartmann described a recursive descent parser with built-in error recovery for an LL(1) grammar, and showed how the error recovery scheme could be generated directly from the grammar, expressed in the form of syntax graphs (see, for example, [WIR76]). Hartmann's method has been analysed by Pemberton [PEM80], who suggested a number of improvements to the original formulation. As was shown by Wirth [WIR76], one alternative to a recursive descent parser derived from a syntax graph is a table-driven parser, in which all the dependence on the grammar is contained in the driving tables, which are, in this case, a direct representation of the syntax graph.

It is natural to enquire whether Hartmann's error recovery scheme may not also be applied to table-driven parsers, and further, if this application may not be generated automatically, by an extension of Wirth's parser generator. The main objective of this paper is to reply affirmatively to these two questions.

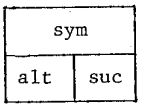
2. TABLE-DRIVEN TOP-DOWN PARSERS

Wirth [WIR76] discussed the problem of parsing programming languages, and showed the equivalence of extended BNF (also known as Regular Right Part, or RRP) grammars and syntax graphs and derived recursive descent and table-driven top-down parsers directly from the grammar. In practical terms, the table-driven parsers offer the advantage of more easily being able to adapt to changes in the grammar, since all the grammar-dependence is concentrated in tables, parametrized for each grammar, whereas for a recursive descent parser a change in the program structure is necessary. Thus, for a parser generator, the table-driven form is clearly superior. In his book, Wirth suggested a concrete representation of the syntax tables and presented a recursive algorithm for the parser, which traverses the syntax graphs, matching the input to the grammar. Setzer [SET79] has shown that this recursive algorithm can be transformed into an iterative one provided we introduce a stack for recording which non-terminals of the grammar are currently active. The two formulations of the parsing algorithm are completely equivalent and use the same representation of the syntax tables, as described in [WIR76] and briefly summarised here.

Each element in the syntax graph is represented by a structure of type node where, using Pascal as a definition language, we have

```
type pointer=↑node;
     node   = record suc,alt:pointer;
                     case terminal:boolean of
                          true:(tsym:termsymbol);
                          false:(nsym:pointer)
              end;
```

where termsymbol is an enumeration type corresponding to the terminal symbols of the grammar. We may represent the node graphically by a box with three fields: a value sym and two pointers, suc and alt, which point, respectively, at the successor element or at a list of alternative elements.



(Note that sym has two variants, tsym and nsym).

The elements in the syntax graphs are either terminal symbols, or non-terminal symbols. A terminal symbol is represented by its value, whilst a non-terminal symbol is represented by (a pointer to) a syntax graph.

It is easily shown that the three constructors of regular expressions (concatenation, alternation and closure) have their correspondents in terms of syntax graphs (sequence, alternation and iteration), and thus in the representation considered here. (see figure 1).

3. HARTMANN'S ERROR RECOVERY SCHEME

The basic idea is to recover from a syntax error by skipping input tokens until we encounter a token which belongs to a set of admissible symbols, called recovery points. After resyncronising the parser with the input at this token, parsing proceeds normally. The crux of the scheme is the choice of the recovery points, which are derived directly from the syntax graphs of the grammar. In a sense which we shall make clearer in the next section, the recovery points consist of terminal symbols which may be derived from elements in the syntax graph which can be reached

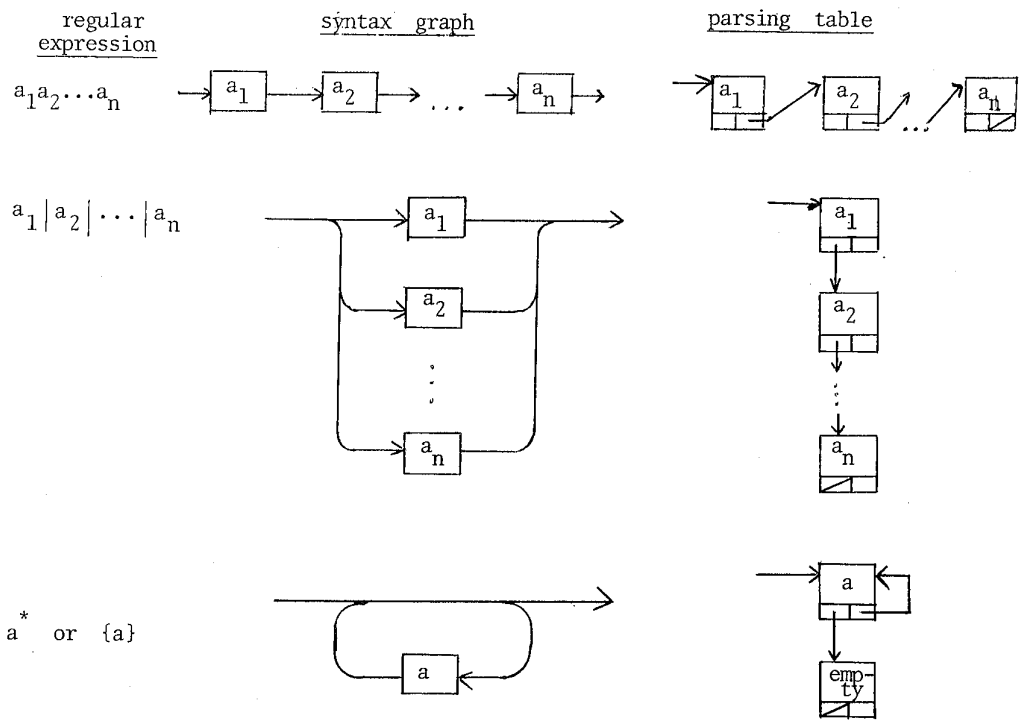


Figure 1: The correspondence between regular expressions, syntax graphs and parsing tables.

from the present position. Some of these points may be obtained statically from the current syntax graph. Others, contained in other syntax graphs, are determined dynamically through the current parser state. These latter recovery points form what Hartmann calls the context.

A syntax error is detected when the next token in the input cannot begin a string derived from the current graph position. In this event, we invoke error recovery as described above.

Hartmann introduced his error recovery scheme in the context of a recursive descent parser, and, in this form, his method was improved by Pemberton [PEM80]. It seems natural to apply the scheme to table-driven parsers derived from syntax graphs. In the next section, we describe the modifications to a table-driven parser needed to incorporate Hartmann's ideas.

4. HARTMANN'S SCHEME APPLIED TO TABLE-DRIVEN PARSERS

Before presenting the table-driven parser with error recovery, it is convenient to introduce some notation in order to facilitate the discussion, and represent clearly our ideas.

Suppose that at a given point in a parse we have reached the element (or component) N. We define T(N), the tail of N, as consisting of those portions of the set of syntax graphs which may be reached from N. We should note that T(N) is composed of $T_{local}(N)$, the local tail of N, consisting of that part of T(N) contained in the current syntax graph, which corresponds to the non-terminal element on the top of the stack, and $T_{global}(N)$, the global tail of N, which consists of the union of the local tails of all the non-terminal elements on the stack. $T_{local}(N)$ is determined statically from the grammar, whereas $T_{global}(N)$ is determined dynamically in the parsing process.

For a given element (or component) N we are interested principally in two sets of tokens associated with N: the director set, and the recovery set. The director set (also called FIRST(N) by Aho and Ullman [AHO77]) contains those tokens which may introduce strings which are derived starting from the component N of the syntax graph. The recovery set consists of the union of the director set of N, with the director sets of all elements (or components) in the tail of N.

Let D(N) be the director set of N, and R(N) the recovery set of N. Thus we have

$$R(N) = D(N) + R(T(N)) \qquad (4.1)$$

where

$$R(T(N)) = R(T_{local}(N)) + R(T_{global}(N)),$$

and + indicates set union.

$R(T_{local}(N))$ is defined to be the union of director sets for all elements contained in the local tail of N,

$$R(T_{local}(N)) = \bigcup_{M \in T_{local}(N)} D(M)$$

Note that

$$C = R(T_{global}(N))$$

is what Hartmann calls the context of the current syntax graph.

Let us consider now the different alternatives for N.

(a) N a terminal element

In this case D(N) is the set consisting of just the corresponding terminal symbol.

(b) N a non-terminal element

D(N) is the set of terminal symbols that begin strings derived from the corresponding non-terminal.

(c) N a sequence

Let N be the sequence $N_1, N_2, \ldots, N_m$ of elements or components. Then $D(N) = D(N_1)$.

(d) N an alternation

N corresponds to a set of alternative sequences, $N_1, N_2, \ldots, N_m$, one of which is possible empty. Then we have

$$D(N) = \bigcup_{1 \le i \le m} D(N_i;T(N)) \qquad (4.2)$$

where ; indicates the sequencing operator. Note that if N_i is empty, then $D(N_i;T(N)) = D(T(N))$, and otherwise $D(N_i;T(N)) = D(N_i)$.

(e) N an iteration

Suppose that the iteration contains a sequence L in the loop. Then we have

$$D(N) = D(L) + D(T(N)) \qquad (4.3)$$

and in this case, since the loop itself can be traversed one or more times, we have

$$R(N) = D(N) + R(L) + R(T(N))$$
$$= R(L) + R(T(N)),$$

because of the definition of D(N).

We are now in a position to explain how Hartmann's method may be incorporated in a table-driven parser. With each terminal and non-terminal element, we associate the director and recovery sets as defined above. With each alternation we associate an alternation element, and with each iteration, an iteration element, as shown in figure 2.

Each alternation and iteration element also has associated with it a director set and a recovery set. In fact, as both Hartmann and Pemberton pointed out, we must associate with an iteration element a third set of terminal symbols, used to exit from an iteration after a syntax error, which we shall denote by E(N), the exit set of N, which is defined by

$$E(N) = R(T(N)) - D(L) \qquad (4.4)$$

The parsing algorithm is now reasonably simple. In a Pascal-like language the main loop may be described as follows:

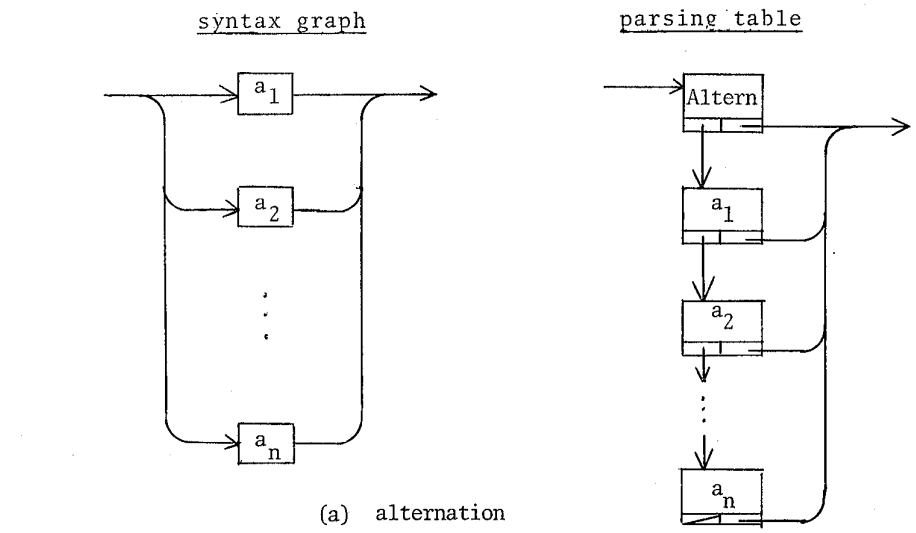
```
while  stack  is not empty and input is not empty
     do while not (nexttoken in D(N))
   (*check for error*)
        do begin while not (nexttoken in R(N))
   (*read input until we find a recovery point*)
                do get (nexttoken);

   (*we may also issue an appropriate error
     message here*)

   case elementtype(N) in (*and resynchronize
                            input*)
   iteration: if nexttoken in E(N) then N:=alt(N)
                                   else N:=suc(N);
   else : while not (nexttoken in D(N))
          do if suc(N)=nil then pop(N)
                           else N := suc(N)
   end(*case*)
end; (*while*)

case elementtype(N) in

terminal: begin get(nexttoken) ; N := suc(N) end;
non-terminal: begin push(N) ; N := nsym(N) end;
alternation: begin N:=alt(N);while not nexttoken
                                 in D(N) do N :=
                                 alt(N) end;
iteration: if nexttoken in D(suc(N))
                                 then N := suc(N)
                                 else N := alt(N)
end (*case*)
end (*main loop*)
```

5. IMPLEMENTATION CONSIDERATIONS

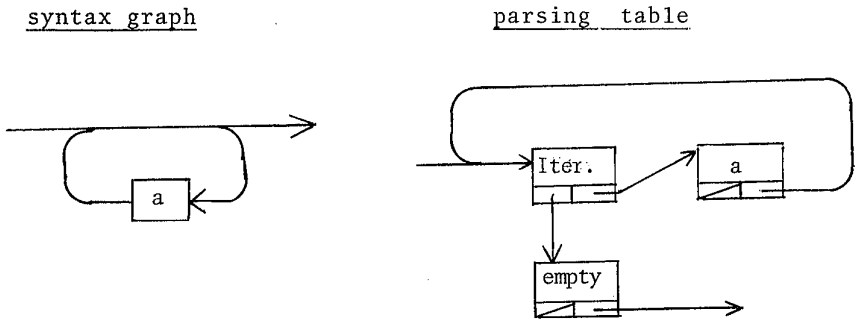
The formulation of the parsing algorithm given in the previous section hides some of the implementation details. These are discussed in this section.

Node representation

The functions suc, alt, elementtype, tsym and nsym are all statically determinable functions of N, the current node. All are thus ideal candidates for table representation. The functions D, R and E have a static part and a dynamic part (see equations (4.1) to (4.4)). The static part can be determined once and for all and inserted in the parsing table, whereas the dynamic part is determined by the parsing algorithm, which unites the two parts whenever this is required. Thus we include in our parsing table the three fields, director, recovery and exit, corresponding to the static parts of D, R and E respectively. Additionally, we need to associate with the iteration elements, and alternation elements with an empty alternative, a boolean field called



(a) alternation



(b) iteration

Figure 2: Parsing table structures for error recovery.

lasti and lasta, respectively, which indicate if $T_{local}(N)$ contains a null path.

A possible alternative representation for the node is as follows:

```
type symbols = set of termsymbol;
     nodetype= (terminal, non-terminal,
                alternation, iteration);

pointer = ↑ node;
node    = record
            suc, alt: pointer
            director, recovery: symbols;
            case elementtype: nodetype of
            terminal: (tsym: termsymbol);
            non-terminal: (nsym : pointer);
```

```
iteration:(exit : symbols;
           lasti:boolean);
alternation : (lista : boolean)
end
```

More efficient use of space can be achieved by noting that the director sets of terminal elements contain only one member, and that the director sets of all instances of the same non-terminal symbol are equal.

The director, recovery and exit sets

As we have noted above, these sets have a static component and (possibly) a dynamic component (see equations (4.1) to (4.4)). The dynamic components may be written

$$D(T_{global}(N)) \text{ and } R(T_{global}(N))$$

and may be obtained as follows:

$$D(T_{global}(N)) = D(M)$$

where M is the successor element (or component) of the non-terminal element on top of the stack. If there is no such successor element, then we look at the previous stacktop, and so on, until we find one.

We have already shown that $R(T_{global}(N))$ (Hartmann's context) is obtained by uniting $R(T(M'))$, where M' is the non-terminal element on top of the stack, with the previous value of the global context.

Note that in both cases, we should stack the old values of $D(T_{global})$ and $R(T_{global})$ when we stack a non-terminal element, in order to restore their values when this non-terminal element is popped.

Thus, whenever we need the value of $R(T(N))$, we must unite $R(T_{local}(N))$ to the current context. Similarly, when we need the value of $D(T(N))$, we must determine first if $T_{local}(N)$ contains a null path. If so, $D(T_{global}(N))$ must be united to $D(T_{local}(N))$.

Semantic actions

In order to build a compiler, the parser must produce output, in what are known as semantic actions. This can be done by introducing a new type of element, of nodetype semantic, which has two fields, suc and action. Action may be the ordinal number of a semantic routine. Alternatively we may add an action field to each of the other kinds of element. Or we may do both of the previous alternatives. One very interesting idea is to generate the semantic elements automatically by extending the grammar to include translation actions, thus obtaining a syntax directed translation scheme (see Barrett and Couch [BAR79]).

Sets

A natural way of implementing sets is as bit vectors. High level languages such as Pascal [JEN75] and several of its derivatives implement sets as basic constructs, and these may be used for the data structures described in this paper.

6. AUTOMATIC CONSTRUCTION OF THE EXTENDED PARSING TABLES

Wirth [WIR76] suggested an algorithm to generate parsing tables in the representation of section 2 using as input an extended BNF form of the grammar. It is a straightforward task to extend his algorithm to generate the additional information needed to implement the scheme we have described. Basically the additional steps needed to reach our goal include.

(a) generation of the alteration and iteration elements
(b) determination of the director sets of the non-terminal elements. Algorithms for this are well known. See, for example, Aho & Ullman [AHO77]
(c) determination of the (local) recovery sets. This is best done by a backwards traversal of each syntax graph, since, as we have noticed,

$$R(T_{local}(N)) = \bigcup_{M \in T_{local}(N)} D(M)$$

and evidently the recovery set of a null graph is empty.
(d) determination of alternation elements with an empty alternative and no local successor
(e) determination of iteration elements with no local successor

Working in groups of three or four students, the above extensions were successfully incorporated in Wirth's algorithm as a class project in a master's level compiler construction course at PUC/RJ.

7. CONCLUSIONS

We have shown how Hartmann's error recovery scheme, which Pemberton has called definitive for recursive descent parsers, may be simply extended to table-driven parsers. The ease of this extension is due to the simplicity of the concepts involved, both of syntax graphs, and of recovery points. The resulting parser is being used in a compiler currently in development at PUC/RJ for the Edison Language [BH81,STA82]. We have also indicate how the additional error recovery data structures may be automatically generated from the grammar, giving a measure of the work involved. Future work will include the development of a general parser generator based on the ideas presented here.

The authors wish to acknowledge the financial support of the Brazilian government agencies Financiadora de Estudos e Projetos (FINEP) and Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq).

8. REFERENCES

[AHO77] Aho,A.H. and Ullman,J.D., *Principles of Compiler Design*, Addison Wesley, Reading, Massachussetts, 1977.

[BAR79] Barrett,W.A. and Couch,J.D., *Compiler Construction:Theory and Practice*, Science Research Associates, 1979.

[BH81] Brinch Hansen,P., "Edison-A Multiprocessor Language", *Software-Practice and Experience, 11* (4), 325-361, 1981.

[HAR77] Hartmann,A.C., *A Concurrent Pascal Compiler for Minicomputers*, Springer-Verlag, Berlin 1977.

[JEN75] Jensen,K. and Wirth,N. *Pascal User Manual and Report*, Springer-Verlag, New York, 1975.

[PEM80] Pemberton,S., "Comments on an Error-recovery Scheme by Hartmann", *Software-Practice and Experience, 10* (3), 231-240.

[SET79] Setzer,V.W., "Non-recursive Top-down Syntax Analysis", *Software-Practice and Experience, 9* (3), 237-245, 1979.

[STA82] Stanton,M.A., et al., "Projeto de um compilador portátil para a linguagem Edison (The design of a portable compiler for the Edison Language)", Annals of the XV National Informatics Congress, Rio de Janeiro, Brazil, October 1982. (in Portuguese).

[WIR76] Wirth,N., *Algorithms + Data Structures = Programs*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.